

# Problem Solving and Agile Software Development

## agile adjective

ag·ile ('a-jəl « -jī(-ə)l «

Synonyms of *agile* >

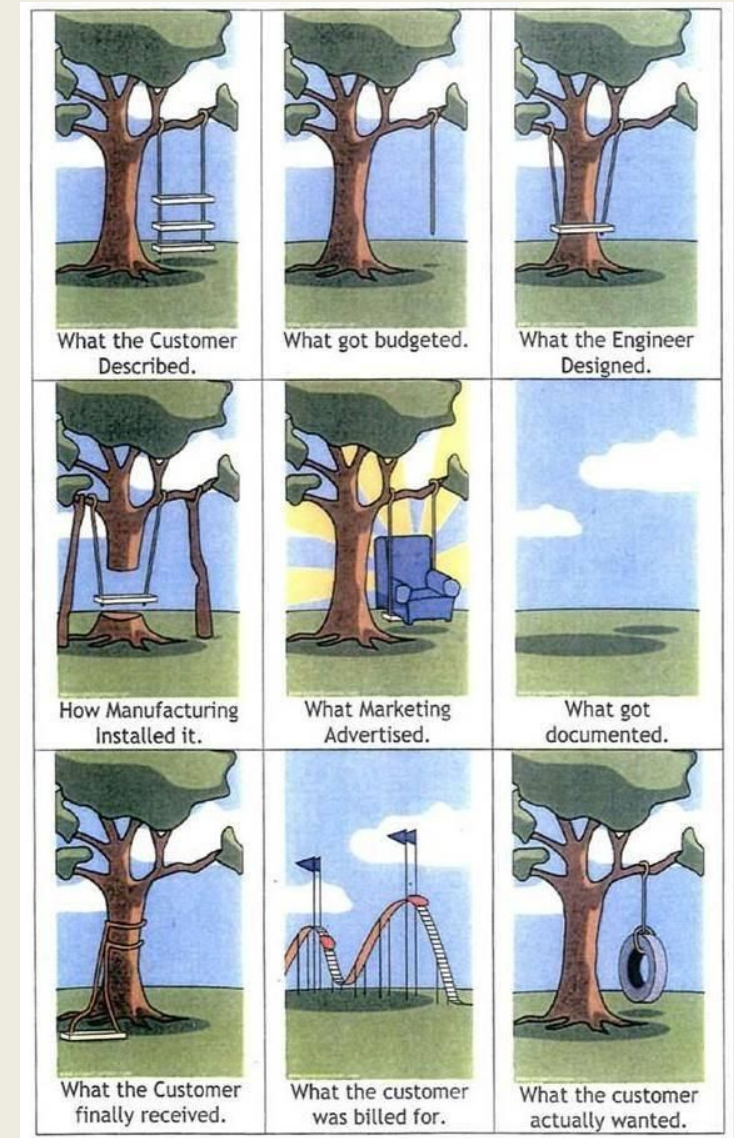
1 : marked by ready ability to move with quick easy grace

| an *agile* dancer

2 : having a quick resourceful and adaptable character

| an *agile* mind

• **agilely** ('a-jəl(l)-lē « -jī(-ə)(l)-lē adverb



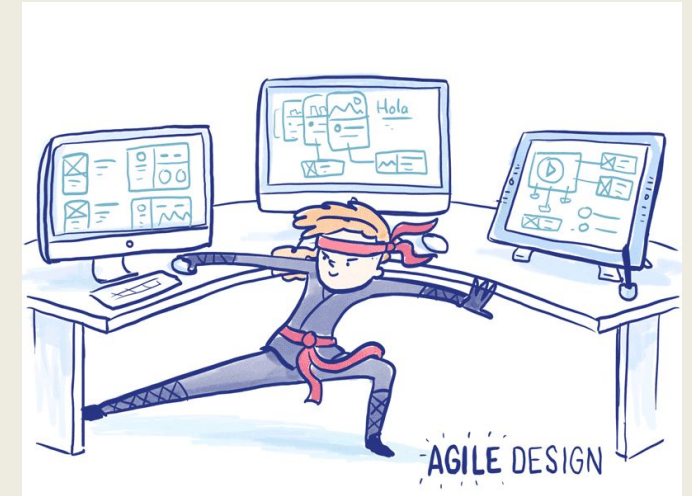
# Agile

- We will approach developing software with an agile mindset.
- This is to guide you on how to create and respond to change and how to deal with uncertainty.
- This will introduce you to how industry approaches building software.

## What is Agile Software Development?

Agile is like Planning a Road Trip with Friends

- Goal/Destination: Agree on where you're heading.
- Flexible Plans: Decide details as you go.
- Team Collaboration: Everyone contributes ideas.
- Regular Check-ins: Stop often, adjust route if needed.
- Deliver in Pieces: Enjoy one stop before moving to the next.



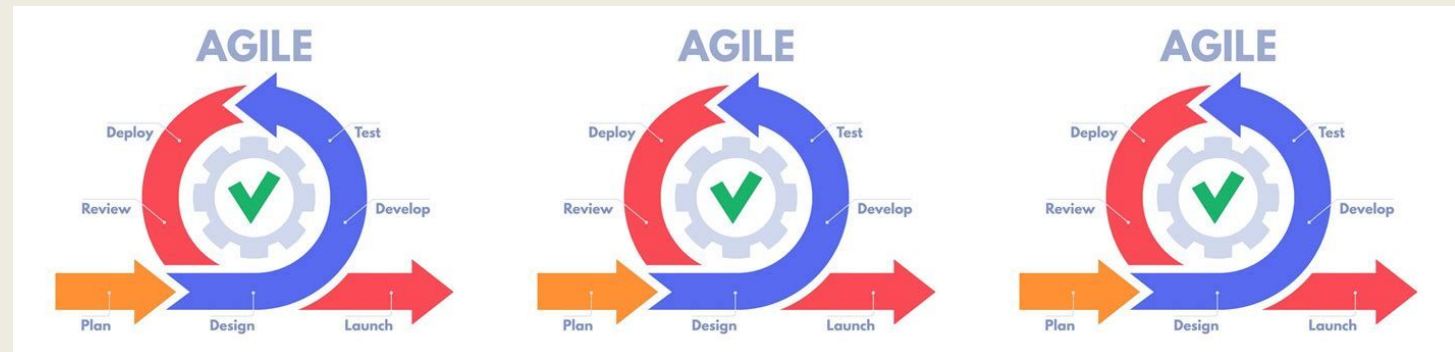
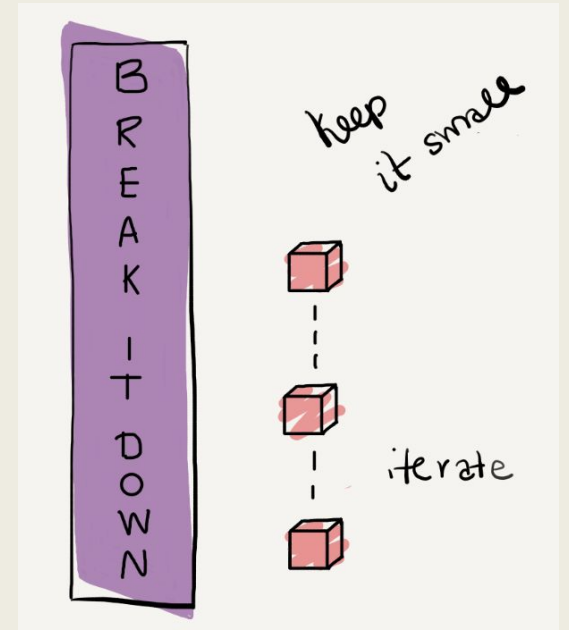
Agile is about flexibility, feedback, and small steps — just like a trip with friends.

# Problem Solving and Software Development

Approach problem solving using Agile Software Development concepts to help you design and implement a solution that is maintainable, testable and extendable.

You will

1. Design:
  - a. Analyze User Stories and Acceptance Criteria (requirements)
  - b. Break the problem into smaller tasks or into classes
  - c. Create test cases for tasks (methods)
  - d. Design Algorithms in Pseudocode for Tasks (Methods)
2. Develop and Unit Test
  - a. Unit test as you implement code
  - b. Create quality code
3. Complete final testing
4. Reflect and Improve



# User Stories and Acceptance Criteria

User Stories are what the user needs and acceptance criteria is how we know it works.

- A short sentence that describes a feature from the user's perspective.
- Format: As a [type of user], I want [some goal] so that [reason/value].

Example (Library App):

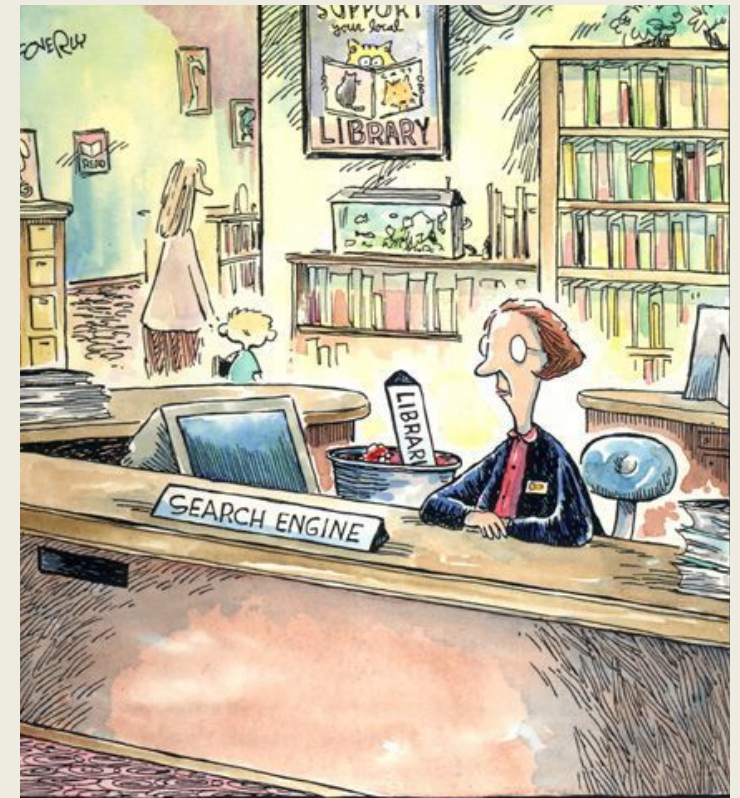
As a librarian, I want to add a book so that it is stored on the shelf.

Acceptance Criteria

- A checklist of conditions that define when the story is “done.”
- Written as clear, testable statements.
- Helps guide both implementation and testing.

Example (for “Add a book”):

- If a slot is available, book is placed and message printed.
- If library is full, system prints error and does not crash.
- Books are always placed in row-major order.





# Library App User Stories

**Story 1 Represent a Book:** As a librarian, I want each book to have its details recorded so that I can identify it on the shelf.

**Acceptance Criteria:** Each book shows its title, author, and year. When I look up a book, I see its information clearly formatted.

**Story 2 Create a Library with Shelves:** As a librarian, I want to set up shelves with space for books so that I know how many I can store.

**Acceptance Criteria:** The library has a name, shelves, and spaces per shelf. If setup is invalid, it still starts safely.

**Story 3 Add a Book Automatically:** As a librarian, I want to add new books into the next available space so that shelves fill in order.

**Acceptance Criteria:** When I add a book, I'm told where it was placed. If the library is full, I'm told it cannot be added.

**Story 4 Display All Books:** As a library member, I want to see what books are available so that I can browse them.

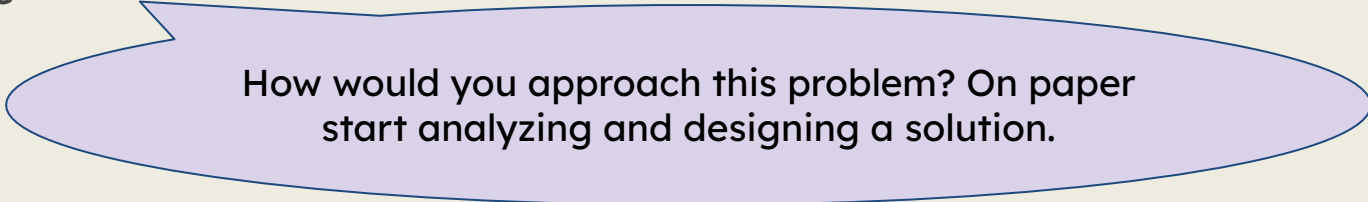
**Acceptance Criteria:** A table lists each book with its location, title, author, and year. Empty spaces not shown.

**Story 5 Count Books Per Shelf:** As a librarian, I want to see how many books are on each shelf so that I can track usage.

**Acceptance Criteria:** A report shows each shelf with the number of books it holds.

**Story 6 Find the Oldest Book:** As a librarian, I want to know the oldest book in the library so that I can feature it.

**Acceptance Criteria:** If no books, I'm told it is empty. Otherwise, I see the oldest year and all books from that year



How would you approach this problem? On paper start analyzing and designing a solution.

# Analyze Requirements and Design Classes

**Design classes around real-world things.**

**One job per class** keeps design clean.

**No copy-paste** reuse code with methods.

## 1. Think in Nouns

- A class usually represents a thing in your program (Book, Student, Library).
- The class holds the data (fields) and the actions (methods) related to that thing

## 2. SRP – Single Responsibility Principle

- “One class, one job.”
- Each class should only do one main thing.
  - Book → stores book details.
  - Library → manages a collection of books.
- If a class tries to do too much, it becomes confusing and hard to test.

## 3. DRY – Don’t Repeat Yourself

- “Write it once, use it everywhere.”
- If you see the same code copied in two places, move it into a method or class.
  - Example: Calculating late fee for an overdue library book
- This makes programs shorter, easier to update, and less error-prone.

# Designing the Loan Class

## [UML Class Diagram Tutorial:](#)

- means private
- + mean public

**Abstract Data Type (ADT):** A class is called an abstract data type - the use of the class is separated from the implementation.

When you create a class, you're creating a new **data type**

- Class contract is the signatures of public and private
  - variables and data types
  - constructors
  - methods parameters and return values

Loan	
-annualInterestRate: double	The annual interest rate of the loan (default: 2.5).
-numberOfYears: int	The number of years for the loan (default: 1)
-loanAmount: double	The loan amount (default: 1000).
-loanDate: Date	The date this loan was created.
+Loan()	Constructs a default Loan object.
+Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double)	Constructs a loan with specified interest rate, years, and loan amount.
+getAnnualInterestRate(): double	Returns the annual interest rate of this loan.
+getNumberOfYears(): int	Returns the number of the years of this loan.
+getLoanAmount(): double	Returns the amount of this loan.
+getLoanDate(): Date	Returns the date of the creation of this loan.
+setAnnualInterestRate(annualInterestRate: double): void	Sets a new annual interest rate to this loan.
+setNumberOfYears(numberOfYears: int): void	Sets a new number of years to this loan.
+setLoanAmount(loanAmount: double): void	Sets a new amount to this loan.
+getMonthlyPayment(): double	Returns the monthly payment of this loan.
+getTotalPayment(): double	Returns the total payment of this loan.

# Developer View

User Stories focuses on customer experience (“I want to see... I’m told when...”)

Developer now focuses on implementation details (arrays, methods, return values).

Both are needed: one drives requirements, the other guides coding.

## **Customer View (plain language)**

### **Story 1: Represent a Book**

As a librarian, I want each book to have its details recorded so that I can identify it on the shelf.

### **Acceptance Criteria:**

Each book shows its title, author, and year. When I look up a book, I see its information clearly formatted.

## **Developer View (technical detail)**

Book class with private fields title, author, year.

Constructor sets fields; getters retrieve values.

toString() formats "Title" by Author (Year).



# Design Classes

## UML Class Diagram Design

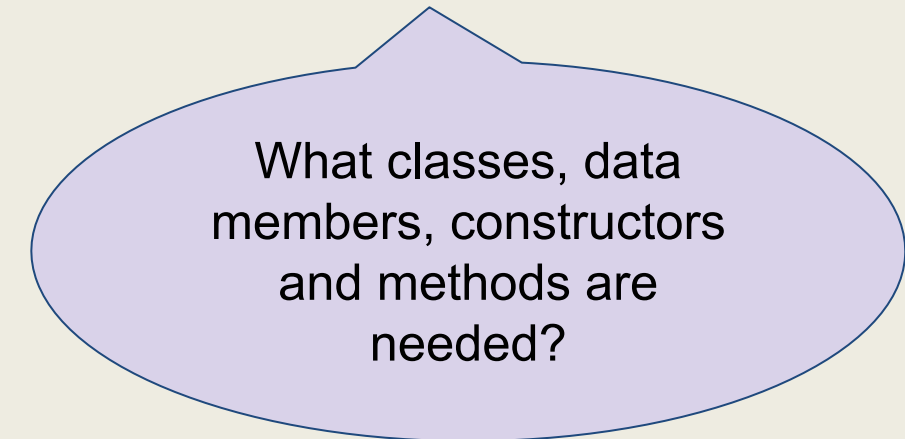
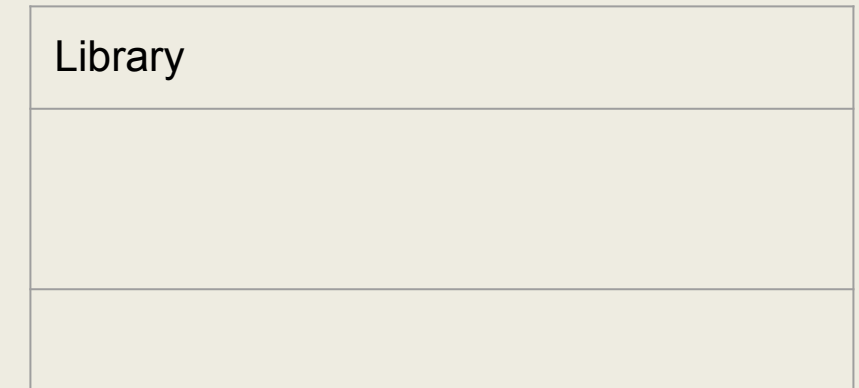
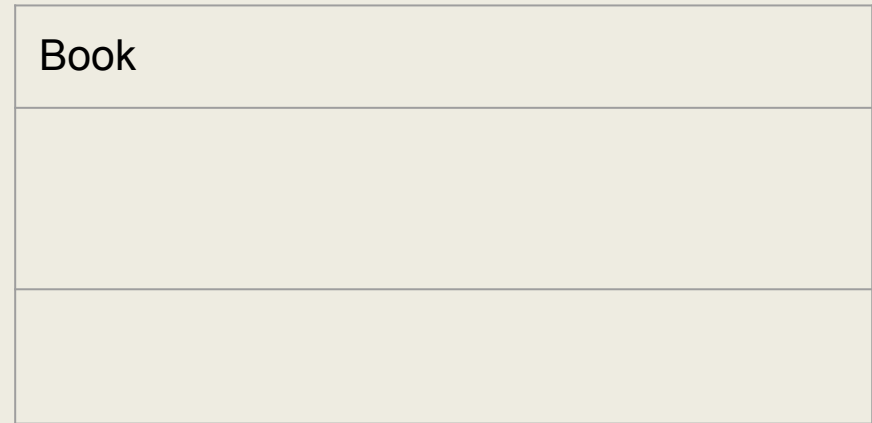
**SRP:** Each class has ONE job

- Book handles details, Library handles management.

**DRY:** Don't Repeat Yourself

- Put shared logic in one place (e.g., Book.toString()).

Refer back to [library app user stories](#) and approach from a developer view



# Book Class Design

**Story 1 Represent a Book:** As a librarian, I want each book to have its details recorded so that I can identify it on the shelf.


**Acceptance Criteria:** Each book shows its title, author, and year. When I look up a book, I see its information clearly formatted.

## Developer View

Book class with

- private fields title, author, year
- Constructor sets fields
- getters retrieve values
- toString() formats "Title" by Author (Year)

Constructor doesn't  
have return value



Book (Follows encapsulation)

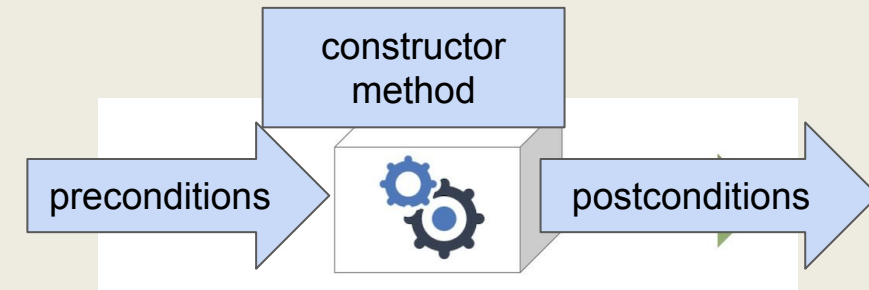
- title: String  
- author: String  
- year: int

+ Book(String:title, String:author, int:year)  
+ getTitle(): String  
+ getAuthor(): String  
+ getYear(): int  
+ toString(): String

# Implement and Unit Test one Class

What is Unit Testing? Unit testing means testing **one small piece of code** (a “unit”) on its own.

- In Java, a “unit” is usually a **single method** or a **single class**.
- Catch problems early before they spread.
- Prove each class works by itself before combining into bigger systems.
- Makes it easier to change or extend code later (confidence).
- **Write small test code in `main`** that:
  - Creates objects with known data.
  - Calls each constructor and method.
  - Prints results and compares to expected.
  - Check: Did the actual output match what you expected?



```
Book book1 = new Book("Unmasking AI", "Joy Buolamwini", 2023);  
// Expected: "Unmasking AI"  
System.out.println("getTitle(): " + book1.getTitle());  
// Expected: "Joy Buolamwini"  
System.out.println("getAuthor(): " + book1.getAuthor());  
// Expected: 2023  
System.out.println("getYear(): " + book1.getYear());  
// Expected: "Unmasking AI" by Joy Buolamwini (2023)  
System.out.println("toString(): " + book1.toString());
```

```
getTitle(): Unmasking AI  
getAuthor(): Joy Buolamwini  
getYear(): 2023  
toString(): "Unmasking AI" by Joy Buolamwini (2023)
```

# Library Class Design

## Developer View

### Library class with

- Library class with field `Book[][]` grid.
- Constructor takes rows and cols;
  - defaults to 1x1 if invalid.
- Method `addBook(Book)` scans row-major order.
  - On success prints confirmation, returns true.
  - If full, prints message and returns false.
- Method `printAll()` loops through `Book[][]`.
  - Prints formatted table of non-null entries.
- Method `printOldest`
  - Prints oldest book(s). If no books, prints "Library is empty." and returns.
- Private Helper Method `countPerShelf()` return the number of books on each shelf

### Library (Follows encapsulation)

- name: String  
- bookShelf: Book[][]  
- numberOfShelves: int  
- shelfCapacity: int  
- currentShelf: int  
- currentSlot: int  
- isFull: boolean

+ Library(name:title, shelves:int, shelfCapacity:int)  
+ getName(): String  
+ addBook(book:Book): boolean  
+ printAllBooks(): void  
+ printOldest(): void  
+ countPerShelf(): int  
- countBooks() : int

How will you implement constructor?

# Constructor

Notice the constructor is checking for invalid data

If invalid data sets array default to [1][1]

```
public Library(String name, int shelves, int shelfCapacity)
{
    if (name == null || name.isEmpty()){
        this.name = "Library";
    } else{
        this.name = name;
    }
    if (shelves <= 0 || shelfCapacity <= 0){
        this.numberOfShelves = 1;
        this.shelfCapacity = 1;
    } else{
        this.numberOfShelves = shelves;
        this.shelfCapacity = shelfCapacity;
    }
    this.totalBookCapacity = numberOfShelves * shelfCapacity;
    this.bookShelf = new Book[numberOfShelves][shelfCapacity];
    this.currentTotalBooks = 0;
    this.currentShelf = 0;
    this.currentSlot = 0;
    this.isFull = false;
}
```

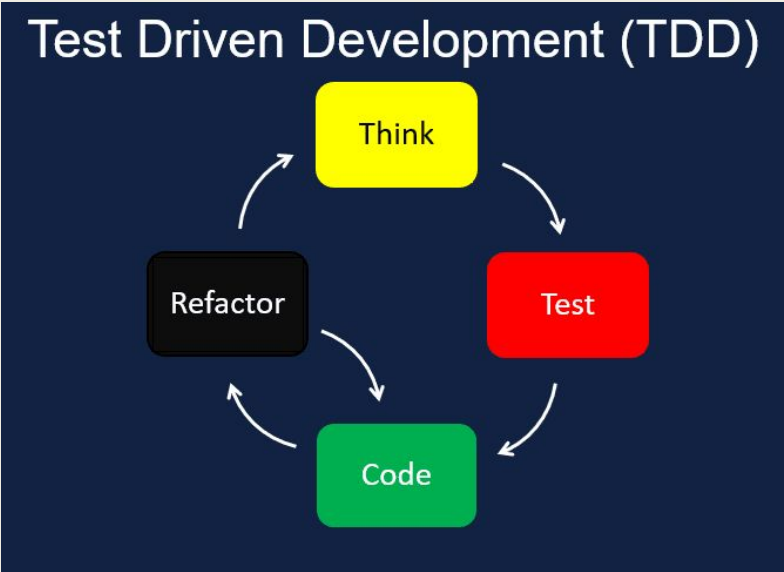


# Use Test Driven Development

When designing more complex algorithms it is helpful to use test driven development to design algorithm

## Test Driven Development (TDD)

- Write test cases first
- Write Code just enough to pass the first test.
- Run it, compare output to your test case output
- Add the next test, update code, and tidy/refactor.
- Repeat.

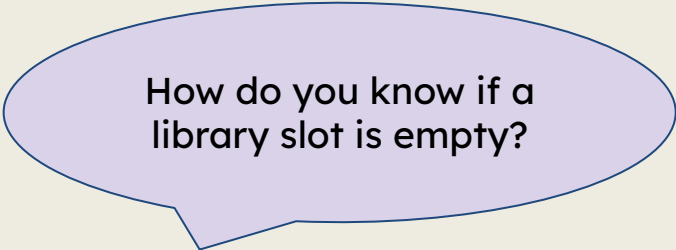


precondition	Action	Expected Result (post condition)
<div>T1: Library has 1 shelf with 2 slots</div> <div>Library library = new Library("Library Test", 1, 2);</div>	<div>Add Book Unmasking AI, Joy Buolamwini, 2023</div> <div>library.addBook(new Book("Unmasking AI", "Joy Buolamwini", 2023));</div>	<div>Displays: Added at shelf 1, slot 1</div> <div>Use debugger 2D Array slot filled at [0][0]</div> <div><div><div><div>library</div><div>bookShelf</div><div>[0]</div><div>[0]</div><div>author</div><div>title</div><div>year</div></div><div>Library (id=23)</div><div>Book[3][] (id=28)</div><div>Book[5] (id=40)</div><div>Book (id=43)</div><div>"Joy Buolamwini" (id=44)</div><div>"Unmasking AI" (id=45)</div><div>2023</div></div></div>
Add more test cases		

# Use Test Driven Development

**Display All Books:** As a library member, I want to see what books are available so that I can browse them.

**Acceptance Criteria:** A table lists each book with its location, title, author, and year. Empty spaces not shown.



precondition Library Has	Action	Expected Result (post condition
T1: 1 shelf with 2 slots	Add Book Unmasking AI, Joy Buolamwini, 2023	Displays: Added at shelf 1, slot 1 2D Array slot filled at [0][0]
T2: 1 shelf 1 of 2 slots filled.	Add Clean Code, Robert C. Martin, 2008	Displays: Added at shelf 1, slot 1 Use debugger: 2D Array slot filled at [0][1]
T3: 1 shelf 2 of 2 slots filled.	Add Refactoring, Martin Fowler, 1999	Displays: Library is full. Couldn't add "Refactoring" by Martin Fowler (1999) Use debugger: 2D Array no change
T4: 1 shelf 2 of 2 slots filled.	Add invalid book library.addBook(null);	Displays: Invalid Book Use debugger: 2D Array no change

# Design Algorithms

- If library has space → book added at correct shelf/slot.
- If library is full → print message, return false.
- If book is null → print message, return false.



precondition Library Has	Action	Expected Result (post condition)	Implement a little code to pass one test.
T1: 1 shelf with 2 slots	Add Book Unmasking AI, Joy Buolamwini, 2023	Displays: Added at shelf 1, slot 1 2D Array slot filled at [0][0]	<code>bookShelf[currentShelf][currentSlot] = book;</code>
T2: 1 shelf 1 of 2 slots filled.	Add Clean Code, Robert C. Martin, 2008	Displays: Added at shelf 1, slot 1 Use debugger: 2D Array slot filled at [0][1]	<code>bookShelf[currentShelf][currentSlot] = book;</code>
T3: 1 shelf 2 of 2 slots filled.	Add Refactoring, Martin Fowler, 1999	Displays: Library is full. Couldn't add "Refactoring" by Martin Fowler (1999) Use debugger: 2D Array no change	<code>if (isFull)</code>
T4: 1 shelf 2 of 2 slots filled.	Add invalid book	Displays: Invalid Book Use debugger: 2D Array no change	<code>if (book == null)</code>

# Refactored

```
public boolean addBook(Book book) {  
    if (book == null) {  
        System.out.println("Invalid book.");  
        return false;  
    }  
    if (isFull) {  
        System.out.println("Library is full. Couldn't add " + book.toString());  
        return false;  
    }  
    bookShelf[currentShelf][currentSlot] = book;  
    System.out.println("Added " + book.toString() + " at shelf " + (currentShelf + 1) + ", slot "  
        + (currentSlot + 1));  
    currentTotalBooks = currentTotalBooks + 1;  
  
    if (currentTotalBooks >= totalBookCapacity) {  
        isFull = true;  
    } else {  
        int nextIndex = currentTotalBooks;  
        currentShelf = nextIndex / shelfCapacity;  
        currentSlot = nextIndex % shelfCapacity;  
    }  
    return true;  
}
```

Check should go first for invalid book

Return if can't place book

check to see if full

Return if can't place book

Return that book was placed



bookShelf	Book[3][] (id=28)
[0]	Book[5] (id=40)
[0]	Book (id=43)
author	"Joy Buolamwini" (id=44)
title	"Unmasking AI" (id=45)
year	2023
[1]	Book (id=26)
author	"Hannah Fry" (id=49)
title	"Hello World" (id=50)
year	2018
[2]	null
[3]	null

addBook method has a single purpose  
There are only a few distinct outcomes:

1. Invalid input: book == null.
  - We can't continue, so we stop right away (return false).
2. No empty slot available
  - The library is full, so there's no point in continuing; return immediately (return false).
3. Success: found an empty slot.
  - Place the book, print a confirmation, return true.

Each of these conditions is self-contained.  
Using early returns here keeps the method simple and readable.

# Quality Software Practices

- Quality software is clear, maintainable, testable code.
- Performance means avoid wasted work
- Our rules are about clarity and predictability:
  -  Use multiple returns & early returns for clarity.
  -  Avoid infinite loops, `System.exit()`, `break`, and `continue` for maintainability.

## Multiple & Early Returns

- Use **early returns** for *simple guard checks* that fail fast (invalid input, full array, null values).
  - a. Example: In `addBook`, return immediately if the book is null or the library is full.
- Avoid them if the method is long, has complex state to manage, or requires cleanup code. They can cause problems if:
  - A method is very long and has many different return points → it becomes hard to see *all the ways out*.
  - Consistency matters — for example, in constructors you often want **one exit point** at the end for clarity.

## No Infinite Loops

- Why: A loop without an end condition can freeze your program.
- Quality impact: Users can't trust the software if it hangs.
- Performance impact: Wastes CPU cycles and memory.

## No break or continue

- Why: These jump out of the middle of a loop and make code harder to follow.
- Quality impact: Future readers (including you!) may miss hidden exits.
- Alternative: Use clear conditions (if checks, early returns) to control flow.



# To Early Return or Not

```
public String readFirstLine(String fileName) {
    try {
        Scanner in = new Scanner(new File(fileName));

        if (!in.hasNextLine()) {
            return null; // EARLY RETURN 1
        }

        String line = in.nextLine();
        if (line.isEmpty()) {
            return null; // EARLY RETURN 2
        }

        return line; // EARLY RETURN 3
    } catch (FileNotFoundException e) {
        return null; // EARLY RETURN 4
    }
}
```

## Problem:

- There are four different return paths scattered across the method.
- Easy to miss cleanup: the Scanner is never closed if an early return happens!
- Harder to read, because you have to scan the whole method to see all the exits.

## Better approach: One return at the end

Here we gather results into a variable and **ensure cleanup happens** before returning.

```
public String readFirstLine(String fileName) {
    String result = null; // default if nothing works
    Scanner in = null;
    try {
        in = new Scanner(new File(fileName));

        if (in.hasNextLine()) {
            String line = in.nextLine();
            if (!line.isEmpty()) {
                result = line;
            }
        }
    } catch (FileNotFoundException e) {
        result = null;
    } finally {
        if (in != null) {
            in.close(); // always runs
        }
    }
    return result; // single exit point
}
```

# Implementing Code Guide for this Class

Why? Read [Code Conventions for the Java Programming Language: 1. Introduction](#)

We will follow the following when implementing code

- [Naming Conventions - Java](#)
- [Braces are used with if, else, for, do and while statements, even when the body is empty or contains only a single statement.](#)
- [One declaration per line since it encourages commenting](#)
- [Put declarations only at the beginning of blocks.](#)
- Do not use ternary operator for this class. While the ternary operator is useful for simple conditions, it can make code harder to read if overused or used with complex logic. In such cases, traditional if-else statements may be clearer

// Use this

```
String resolvedName;  
if (name == null || name.isEmpty()) {  
    resolvedName = "Library";  
} else {  
    resolvedName = name;  
}
```

// do not use

```
String resolvedName = (name == null || name.isEmpty()) ? "Library" : name;
```

# Code Design and Layout

When implementing remember

- [Software Design Principles Don't Repeat Yourself \(DRY\) and Keep It Simple Silly \(KISS\)](#)
- [Single Responsibility Principle \(SRP\)](#)

You will put all code information in one file. When you define a public class in a Java file it must match the file name.

1. Put driver class with main method first. This class is accessible by other classes, and it acts as the main entry point for your program if it contains the main method. Java allows only one public class per file to maintain a clear, organized structure.
2. Follow with user defined class and they are not marked as public, which means it has package-private access by default. This class to be used only by ClassDriverTestClass
  - a. variables (implement encapsulation)
  - b. constructor
  - c. methods

We will not be utilizing packages or separate files for your user defined classes.

```
public class ClassDriverTestClass
{
    public static void main(String[] args)
    {
        }//end main

    public static void method()
    {
        }//end method
} //end class

class ClassName {
    private double instanceVar1;

    /** Constructor
    public ClassName() {
    }

    /** Overload Constructor*/
    public ClassName(double newInstanceVar1) {
        instanceVar1 = newInstanceVar1;
    }

    /** class method*/
    double method1() {
        return instanceVar1;
    }
} // end className

ADD MORE CLASSES
```

# Avoid Hardcoding

Avoid [hard coding](#) data

Hard-coded numbers also called “magic numbers.”

They make code hard to read, hard to update, and easy to break.

```
if (booksOnShelf > 5) { // Why 5?  
    System.out.println("Shelf full!");  
}
```

If shelf size changes to 10, you must hunt down every 5 and update it.

Easy to miss one, causing bugs.

Instead, use constants, variables, or parameters so your code is clear, maintainable, and adaptable.

```
private static final int MAX_SHELF_CAPACITY = 5;  
if (booksOnShelf > MAX_SHELF_CAPACITY) { ... }
```

# Readable Documented Code

Documenting code with comments.

- Initial comment above code
- JavaDoc comments above methods
- `//` single line comments for complex code

You can [Generate JavaDoc Comments](#) for methods and classes and then edit

Initial comment at top of source file should include name, class and description of what program does.

```
/**
 * Library App Test Driver
 * Contains Book and Library Classes to be tested
 * 1) Set up library
 * 2) Add up to capacity
 * 3) Print table of books
 * 4) Print count per
 * 5) Print oldest book(s)
 *
 */
```

Notice how the built in methods contain comments

`Character.isDigit`

`isDigit(char ch) : boolean - Character`

Determines if the specified character is a digit.

**Parameters:**

`ch` the character to be tested.

**Returns:**

true if the character is a digit; false otherwise.

See Also:

Comments above method you create should contain summary of what method does and describe the parameters and return value

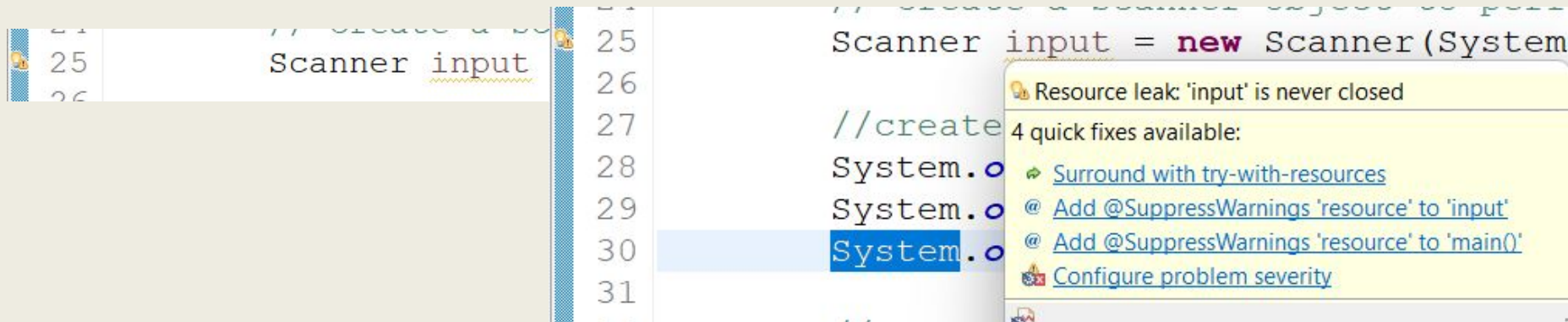
```
/**
 * Constructs a library with the given name and shelf dimensions.
 * If dimensions are invalid default of 1x1 is used.
 * @param name      library name
 * @param shelves   number of shelves (rows)
 * @param shelfCapacity number of slots per shelf (columns)
 */
public Library(String name, int shelves, int shelfCapacity)
```



# Eliminate Warnings

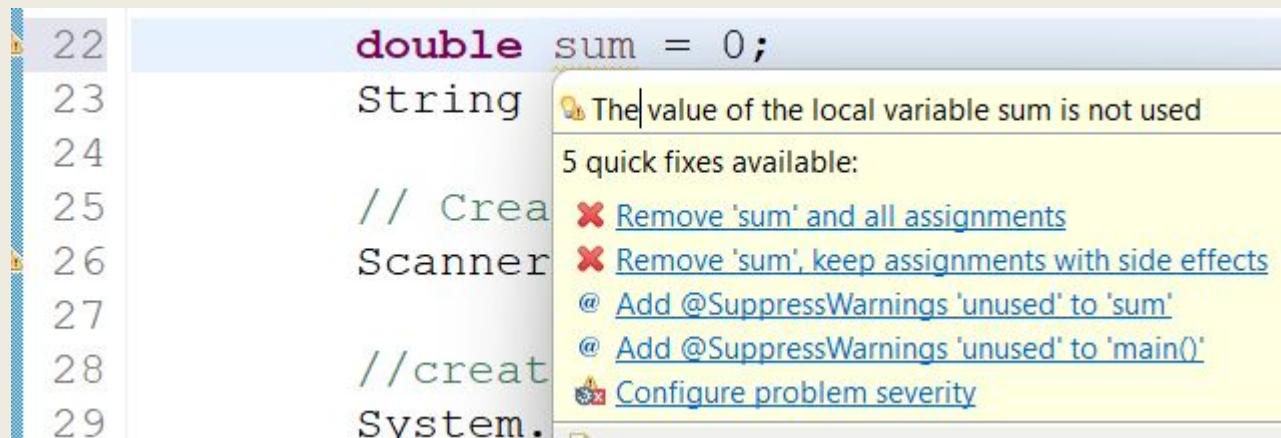
Warnings are given to improve the quality of your code.

- Look for the warnings and the message of suggestions on how to fix.
- Use what you have learned in class to fix. Get help if you aren't sure how to fix it.



This screenshot shows a code editor with a warning icon on line 25. The code is: `Scanner input = new Scanner(System.in);`. A tooltip is displayed over the warning, stating "Resource leak: 'input' is never closed". It lists four quick fixes: "Surround with try-with-resources", "Add @SuppressWarnings 'resource' to 'input'", "Add @SuppressWarnings 'resource' to 'main()'", and "Configure problem severity".

```
25 Scanner input = new Scanner(System.in);
26
27 //create
28 System.out.println("Enter a number:");
29 System.out.println("Enter a number:");
30 System.out.println("Enter a number:");
31
```



This screenshot shows a code editor with a warning icon on line 22. The code is: `double sum = 0;`. A tooltip is displayed over the warning, stating "The value of the local variable sum is not used". It lists five quick fixes: "Remove 'sum' and all assignments", "Remove 'sum', keep assignments with side effects", "Add @SuppressWarnings 'unused' to 'sum'", "Add @SuppressWarnings 'unused' to 'main()'", and "Configure problem severity".

```
22 double sum = 0;
23 String
24
25 // Create
26 Scanner
27
28 //create
29 System.out.println("Enter a number:");
```

# Learnings and Reflection

What is a Retrospective? Reflect after an iteration to improve on the next.

In industry we would celebrate successes, discuss lessons learned, and any necessary changes to improve.

Reflecting and documenting learnings is a valuable process, so you can improve.



*Celebrate*

# Plan Your Time

Look at your tasks and plan dates you will work on each task - [Project Planning](#)

SUNDAY	MONDAY	TUESDAY	WEDNESDAY	THURSDAY	FRIDAY	SATURDAY
					1 List tasks to code and unit test	2
3 List Tasks to code and unit test	4	5 Finish coding and testing	6	7 Finish coding and testing	8	9 Review assignment details and if complete submit
10	11 DUE!	12	13	14	15	16

[What you can do using project planning on GitHub - DEV Community](#)